Compact hardware Liquid State Machines on FPGA for real-time speech recognition

Benjamin Schrauwen [*,1], Michiel D'Haene [2],
David Verstraeten [2], Jan Van Campenhout

Department of Electronics and Information Systems, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium

Abstract

Hardware implementations of Spiking Neural Networks are numerous because they are well suited for implementation in digital and analog hardware, and outperform classic neural networks. This work presents an application driven digital hardware exploration where we implement real-time, isolated digit speech recognition using a Liquid State Machine. The Liquid State Machine is a recurrent neural network of spiking neurons where only the output layer is trained. First we test two existing hardware architectures which we improve and extent, but that appear to be too fast and thus area consuming for this application. Next, we present a scalable, serialized architecture that allows a very compact implementation of spiking neural networks that is still fast enough for real-time processing. All architectures support leaky integrate-and-fire membranes with exponential synaptic models. This work shows that there is actually a large hardware design space of Spiking Neural Network hardware that can be explored. Existing architectures only spanned part of it.

Key words:
Liquid State Machine, Spiking Neural Network, FPGA, Speech recognition

* Corresponding author.
Email address: Benjamin.Schrauwen@UGent.be (Benjamin Schrauwen).
URL: http://www.elis.ugent.be/SNN (Benjamin Schrauwen).

[1] This work is partially funded by the FWO Flanders project G.0317.05.

[2] David Verstraeten and Michiel D'Haene are sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

1 Introduction

Spiking Neural Networks (SNNs), neural network models that use spikes to communicate, have (1) been shown theoretically [1] and practically [3] [2,3] to computationally outperform analog neural networks, (2) are biologically more plausible, (3) have an intrinsic temporal nature that can be used to solve temporal problems, and (4) are well suited be implemented on digital and analog hardware. SNNs have been applied with success to several applications such as face detection [4], lipreading [2], speech recognition [5] and speaker identification [6], autonomous robot control [7,8] and several UCI benchmarks [9].

The main drawback of SNNs is that they are difficult to train in a supervised fashion mainly because the hard thresholding that is present in all simple spiking neuron models makes the calculation of gradients very prone to errors which deteriorate the learning rule's performance [9,10,3]. One way to circumvent this is by using fixed parameters. This is what is embodied by the Liquid State Machine (LSM) concept [11] (which is conceptually identical to Echo State Networks [12] and which are generally termed Reservoir Computing [13]). Here a recurrent network of spiking neurons is constructed where all the network parameters (interconnection, weights, delays, ...) are fixed and randomly chosen. This network, the so called liquid or reservoir, typically exhibits complex non-linear dynamics in its high-dimensional internal state. This state is excited by the network input, and is expected to capture and expose the relevant information embedded in the latter. As is the case with kernel methods, it is possible to extract this information by processing these network states with simple linear techniques to obtain the actual regression or classification output.

Neural networks can be, and are often, implemented through simulation on sequential computers. This approach obviously limits the speed of operation of the network. Direct parallel hardware implementations of neural networks have been a fruitful research area due to their intrinsic parallel nature which allows very large speedups compared to sequential implementations. Several general overview publications on digital hardware implementations of neural networks have been published [14–18], some of them specific to implementations on Field Programmable Gate Arrays (FPGA) [19–21], and specific for digital spiking neural networks [22–25], or neuromorphic analog VLSI [26]. In this work we focus on digital implementations on FPGAs, which are reconfigurable digital hardware components, because they offer an intermediate between the programmability of classic processors, and the parallel nature and high speed

[3] The linearly non-separable XOR function can be represented by a single spiking neuron.

Table 1
Taxonomy of hardware implementations of SNNs

	time-step based			event based
	FPGA	ASIC	MPU	FPGA
multiple PEs				
no timesharing	[7,30,31,33]		[29]	
timesharing	[32,39,43,42]	[34,35]		[37,41]
single PE				
timesharing	[40]	[36,38]	[28]	

of ASICs. Also, FPGAs allow a much faster implementation cycle (minutes instead of months) and for a 'small' number of chips (currently less than 100000!) FPGAs are cheaper than ASICs.

The main reasons for looking at spiking neuron hardware implementations is that: information is represented in time instead of in space allowing to better use the intrinsic high speed of digital hardware, a spiking communication channel can carry more information than an analog (rate coded) communication channel [27] allowing for lower bandwidth interconnection links in hardware, due to the on/off natur,e no multiplications have to be performed at the input of a neuron, and a broad range of architectures is possible spanning a large continuum in design space which allows the user to make an area/speed trade-off depending on the application. Several digital implementations of SNNs have already been published, ranging from very simple neurons implemented on 8-bit micro-controllers [28,29] to somewhat larger FPGA based systems [7,30–33] and finally very large systems comprising several FPGAs or even ASICs [34–43]. A taxonomy of the architectures presented in these publication is presented in Table 1. We subdivide the designs according to simulation principle, hardware platform, the number of Processing Elements (PEs) and if they use time-sharing (multiple neurons are processed on the same hardware).

The LSM architecture has many properties that are advantageous to hardware implementations: weights are fixed and chosen at random (high weight quantization noise can be taken into account a priori), interconnection topology can be very sparse and even 'small world' (many local connections, few global ones) which allows easy wireability that matches the intrinsic FPGA wiring capabilities well. Also, as for many LSM applications quite large networks of spiking neurons (up to 1000) need to be simulated with hard real-time constraints which is difficult in software or when using event-based simulation

techniques, multiple outputs can be generated from the same reservoir [4] allowing a generic hardware reservoir component that can operate on different applications and with multiple outputs, and due to the intrinsic LSM robustness even faulty FPGAs can be used or a higher yield of ASICs is possible [44].

Recently a very convincing engineering application for the Liquid State Machine was presented: isolated spoken digit recognition [5]. When optimally tweaked [45], it can outperform state-of-the-art hidden markov model based recognizers. The system is biologically motivated: a model of the human inner ear is used to pre-process the audio data, next an LSM is constructed with biologically correct settings and interconnection [11], and a simple linear classifier is used to perform the actual classification.

In this paper we present an application oriented design flow for LSM-based hardware implementation. Real-time, single channel speech recognition with the lowest hardware cost is desired. To attain this goal we implement the speech task on two existing hardware architectures for SNNs: a design that processes synapses serially and which uses parallel arithmetic [7,30] and a design that processes the synapses in parallel, but does the arithmetic serially [46,33]. As we will show, these architectures are always much faster than real-time, and thus waste chip area. For all digital implementations in general, a broad range of different architectures is possible with very different properties with respect to chip area, memory usage and computing time. Most of the time the properties are contradictory and therefor a area/time trade-off has to be made. We present a new architecture that uses both serial synapse processing and serial arithmetic. Using this option we are able to process just fast enough for real-time with a very limited amount of hardware. Without much extra hardware cost this design allows to easily scale between a single PE which performs slow serial processing of the neurons to multiple PEs that each process part of the network at increased speed. The design space for hardware SNNs has thus been drastically enlarged. All presented designs were implemented at our lab and run on actual hardware. The LSM-hardware idea (but with threshold logic neurons) was previously already implemented in analog VLSI hardware in [44].

All above mentioned hardware implementations of SNNs only implement very simple Leaky Integrate and Fire (LIF) spiking neurons (or even the simpler Integrate and Fire). This is the simplest first-order or single-compartment spiking neuron model. It was theoretically shown by [47] that second-order models (which have a model for synaptic dynamics) are computationally superior to first-order spiking neurons, and the proof that SNNs are more powerful (in

[4] For example in [6] we showed that is possible to do speech and speaker recognition simultaneously with the same reservoir.

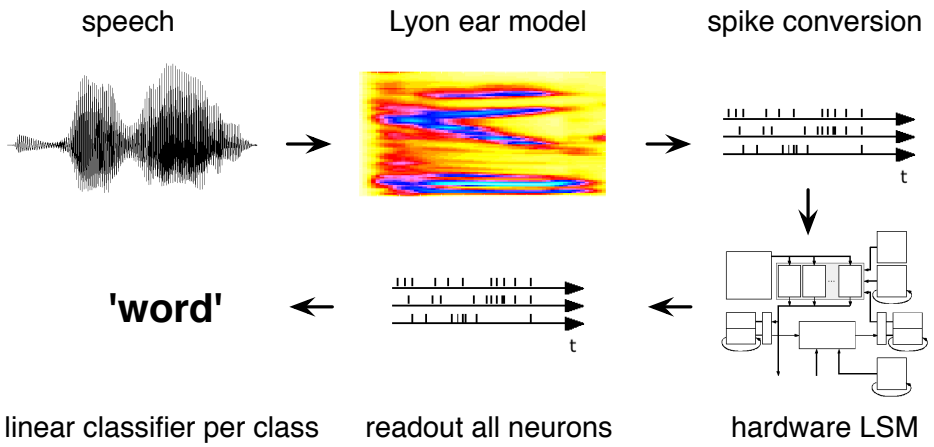

Fig. 1. Overview of the overall hardware speech recognition presented in this work.

the sense of computational efficiency) than ANNs [1] is based on second-order SNNs. It was also shown in [13] that reservoirs with second-order neurons outperform first-order neurons. This is why we chose to include second-order neurons in all implementation architectures discussed in this work.

For all hardware architectures studied in this publication, network topologies can be automatically generated by Matlab code which is part of the Reservoir Computing toolbox [5] presented in [13]. This way, network structures can first be explored in software, and when a good topology is found, it can easily and automatically be exported to a structured hardware description which fits in the an automated design flow.

This contribution is structured as follows. Section 4 presents the class of neuron models that will be implemented in hardware, and Section 5 shows how these models can be efficiently approximated digitally in a time-step based simulation. An overview of the different possible forms of parallelism and interconnection is given in Section 6. Three detailed implementations, each with very specific spatial and temporal characteristics, are presented in Section 7 and Section 8 compares their properties in terms of the important network and neuron parameters. To show the real-world applicability of these systems on an LSM application, Section 9 presents an isolated speech recognition system based on the implementation architectures presented in this work. We conclude and point out future directions of research in Section 10.

2 Application: isolated digit speech recognition

The isolated digit speech recognition application that will be implemented in hardware is organized as follows: a much used subset of the TI46 isolated digit corpus consisting of 10 digits uttered 10 different times by five female speakers was preprocessed using Lyon's passive ear model (a model of the human inner ear)[48]. The multiple channel result of this preprocessing step is converted to spikes using BSA [49], a fast spike coding scheme with a good signal to noise ratio. The resulting spike trains are fed into a randomly generated network of spiking neurons, whose parameter settings are optimized using a Matlab toolbox for RC simulations designed at our lab [6]. The responses of the network (the spikes emitted by the neurons) are converted back to the analog domain using an exponential low-pass filter to mimic the operation of a neuron membrane as described in [50] and then resampled using a time-step of 30 ms. The resulting time series are used as input to 10 linear classifiers, one for each digit, which are trained with a one-shot learning approach based on a pseudo-inverse approach with regularization. Training and testing is done in a 10-fold cross-validation setup. The output of the classifiers is post-processed by taking the temporal mean of each classifier's output and applying winner-take-all to the resulting class-vector. The effect of different weight parameter settings were evaluated, where the performance is measured using the word error rate (WER) , which is simply the fraction of misclassified digits. A reservoir was selected that attained a WER of around 5% when simulated with a software implementation of the hardware model (which is part of the toolbox) which takes the quantization effects into account.

The reservoir consists of 200 spiking neurons, all with the same threshold and reset values (255 and -255 respectively) and both absolute and relative refractoriness. The input weights are randomly chosen as either -0.1 or 0.1 and then scaled by the value of the neuron threshold. The internal weights were normally distributed, multiplied by the threshold and rescaled according to the spectral radius (which is a parameter controlling the network dynamics, but which was originally defined for sigmoidal neural networks). The optimal spectral radius was determined to be 0.1. Each neuron receives 12 input connections, 8 from other neurons in the reservoir and 4 from the input.

[5] The RC toolbox and the VHDL designs are all open source and available on-line at http://rc.elis.ugent.be/ `http://rc.elis.ugent.be/`.

[6] This is an open source research toolbox containing a complete scala of reservoir based techniques. It is available at `http://www.elis.ugent.be/rct`.

3 Hardware oriented RC design flow: RC Matlab toolbox

To generate the hardware in this work, we used the following rough guidelines on how to tackle an engineering problem using hardware Reservoir Computing. To do this, we use the RC toolbox presented in [13] which offers a user-friendly environment to do a thorough exploration of certain areas of the parameter space, and to investigate some optimal parameter settings in a software environment before making the transition to hardware. The following steps are advisable:

- **Generic network and node settings.** Reservoirs are determined by a broad range of topology and neuron parameters that can influence the performance of the networks significantly. Using the RC toolbox, optimal settings for a given task can flexibly be determined and evaluated. Much of the parameter tuning is as yet empirical and dependent on experience with reservoir computing techniques. Experimental indications for good parameters are described in [13] for many reservoir types, and in [51] for Echo State Networks specifically.
- **Readout pipeline.** Once an optimal reservoir is found, the post-processing of the reservoir states can further improve performance. The RC toolbox offers a number of nonlinear and filter operations that can be applied to the output of the regression step. These simple operations can also greatly influence performance, especially for highly temporal regression tasks, such as speech recognition.
- **Generate network with hardware constraints.** All hardware designs discussed in this contribution take up a minimal amount of hardware area if the structure of all the neurons is as regular as possible (i.e. every neuron has the same number of inputs, synapse models and input distribution) , because then only a single controller for the entire network is needed instead of one controller per PE. Using the RC toolbox, reservoirs with a regular structure can be easily constructed, so the user can evaluate the influence of these constraints on the performance.
- **Evaluate node quantization effects.** When the transition is made from software to hardware, calculations change from floating-point to fixed point arithmetic with a tunable precision. This introduces a trade-off between memory and hardware requirements on the one hand, and quantization noise on the other hand. These quantization effects can also be modeled and simulated using the RC toolbox, which allows the user to take these effects into account when designing the reservoir. Simulation models for the hardware neuron model with its specific quantisation effects are available in the toolbox. If necessary, an iterated approach of the first four steps is possible.
- **Automatically export to hardware.** Once the optimal reservoir has been determined, the RC toolbox offers helper functions that allow the reservoir

structure to be exported automatically to a VHDL hardware description (only a small settings file is generated, the hardware architectures themselves are completely generic and automatically adapt to the setting in this file). We have helper functions for the three architectures presented in this work. The hardware description is automatically synthesised to an actual hardware implementation.

4 Neuron models

A multitude of biologically inspired neuron models exist (see [52] for a good overview of different neuron models), ranging from complex compartemental structures using ion channel models, to the very simple integrate-and-fire model. Most of these neuron models can be approximated very well by means of the Spike Response Model (SRM) [53]: a neuron is modeled by a superposition of pre-synaptically induced time-dependent input 'kernels' (pulse response functions or convolutions, not high dimensional feature mappings as in SVMs) and a post-synaptically induced time-dependent depolarization kernel. A slightly simpler model, called SRM_0, is often used; here the time dependence of the kernels is dropped. The membrane potential according to the SRM_0 can be written as:

$$u_i(t) = \sum_{t_i^{(f)} \in \mathcal{F}_i} \eta_0\left(t - t_i^{(f)}\right) + \sum_{j \in \Gamma_i} \sum_{t_j^{(f)} \in \mathcal{F}_j} w_{ij} \epsilon_0\left(t - t_j^{(f)}\right), \tag{1}$$

with u_i denoting the membrane potential of neuron i, w_{ij} denoting the weight of the connection between neuron j and i, Γ_i the set of all the neuron numbers connecting to neuron i, $t_j^{(f)}$ representing the f'th firing time of neuron j and $\mathcal{F}_j$ the set of all these firing times. The membrane response to input spikes or post synaptic potential (PSP) is modeled by $\epsilon_0(t)$, and to spikes generated by the neuron itself (when the threshold is reached) or spike after potential (SAP) by $\eta_0(t)$. The neuron i generates output spikes $t_i^{(f)} \in \mathcal{F}_i$ whenever the membrane potential u_i reaches the threshold ϑ:

$$\mathcal{F}_i = \left\{ t_i^{(f)} \,:\, u_i\left(t_i^{(f)}\right) = \vartheta \text{ and } \frac{\mathrm{d}u_i\left(t_i^{(f)}\right)}{\mathrm{d}t} > 0 \right\}. \tag{2}$$

Spike trains can thus be modeled as either a set of spike times, $t_i^{(f)} \in \mathcal{F}_i$, or as a function of time equal to a sum of Dirac pulses, $s_i(t) = \sum_{t_i^{(f)} \in \mathcal{F}_i} \delta\left(t - t_i^{(f)}\right)$.

In this paper we will focus on neurons that use a Leaky Integrate and Fire

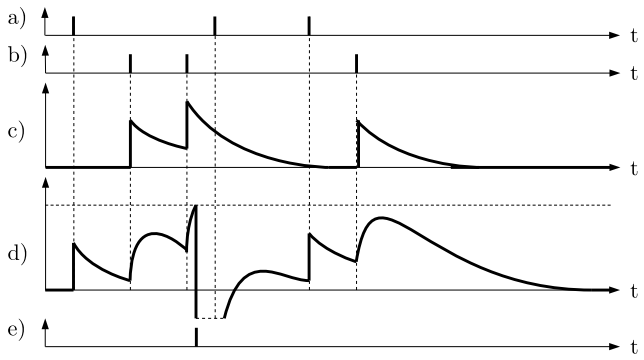

Fig. 2. LIF type membrane with direct incoming connections (first order) and an exponential synapse model (second order response). Trace (a) shows the input spikes directly to the membrane potential, trace (b) shows the spikes to the exponential synapse model, trace (c) shows the potential of the synapse model, trace (d) shows the membrane potential which has an absolute refractory period, and trace (e) shows the output spikes.

(LIF) membrane model. The SAP then becomes

$$\eta_0(t) = -(\vartheta - u_r)\exp\left(-\frac{t}{\tau_m}\right)\mathcal{H}(t), \tag{3}$$

with $\mathcal{H}(t)$ the Heaviside function, u_r the reset potential and τ_m the time constant of the membrane model. We always assume the rest potential to be zero.

The PSP depends on the synaptic current generated by the synapse when a spike is received; different synaptic models can thus be used. When using a LIF membrane model, the PSP $\epsilon_0(t)$ relates to the synaptic current $\alpha(t)$ as

$$\epsilon_0(t) = \int_0^\infty \exp\left(-\frac{s}{\tau_m}\right)\alpha(t-s)\ \mathrm{d}s\,\mathcal{H}(t)\ . \tag{4}$$

We will look at two often used synaptic models: the simplest possible where the synaptic current is modeled by a Dirac δ-pulse, that is, $\alpha(t) = \delta(t)$; and the more realistic model where the post-synaptic current is modeled by an exponentially decaying function with time constant τ_s:

$$\alpha(t) = \exp\left(-\frac{t}{\tau_s}\right)\mathcal{H}(t)\ . \tag{5}$$

The neuron model we will use in this publication thus consists of a LIF membrane model with Dirac and/or exponential synapse models. The SRM for these neurons can be rewritten as:

$$u_i(t) = \int_0^\infty \exp\left(-\frac{s}{\tau_m}\right) \left[\sum_j w_{ij} \sum_f \alpha\left(t - t_j^{(f)} - s\right) - (\vartheta - u_r) \sum_f \delta\left(t - t_i^{(f)} - s\right)\right] \mathrm{d}s\, \mathcal{H}(t) \tag{6}$$

which is the exponential decay of the combination of the input post-synaptic currents and the neuron's own generated pulses used for resetting the neuron whenever it fires. The post-synaptic currents are Dirac pulses or exponential responses to Dirac pulses. Exponentially decaying integrators are thus a key concept in the implementations that will be discussed in the remainder of this publication.

Note that in the previous paragraphs we stated that all the neuron's synapses are of the same type. In what follows we will assume that several synapse models per neuron (type and time constant) are allowed. This is biologically more plausible than only one synaptic time constant, and introduces several different time scales into a single neuron which enhances its temporal computation capabilities.

We also model refractoriness. Refractoriness stands for the period of time after a neuron has fired during which it is less or not sensitive at all to input spikes. This is called relative and absolute refractoriness, respectively. We identify three different types of refractoriness: relative refractoriness where the membrane potential is reset to a value below the rest potential making the neuron less sensitive after it has fired, absolute refractoriness which blocks all input spikes for a short time after the neuron has fired, and a combination of both of these where we first have a period of absolute refractoriness followed by a less sensitive period.

Although interconnection delay is an important parameter we will not implement it because implementing arbitrary delays on digital hardware is complex to implement in the simple implementations we will elaborate on, and consumes a lot of hardware resources. In practice, synaptic delays can be partially approximated by appropriate choices of weights and synaptic time constants. Another way of introducing delay is due to the second-order neuron models, which allow to have delayed firing (a neuron fires several milliseconds after that a spike arrives), which is not possible with first-order models (neurons can only fire when an input spike arrives).

5 Digital approximations of exponential decay

Exponential decay is the key ingredient for the implementation of the LIF-type membranes with Dirac and/or exponential synapse models mentioned above. Even short- and long-term adaptation rules such as Spike Timing Dependent Plasticity [54], Dynamic Synapses [55] and Intrinsic Plasticity [56] are all based on exponential decay. In this section we will further investigate how we can efficiently and compactly implement exponential decay on digital hardware components.

The generic exponentially decaying convolution operation

$$y(t) = \int_0^\infty \exp\left(-\frac{s}{\tau}\right) x(t-s)\ \mathrm{d}s$$

can be approximated in discrete time by

$$y[t] = \sum_{k=0}^{\infty} \exp\left(-\frac{k\Delta}{\tau}\right) x[t-k] = a\, y[t-1] + x[t]$$

where Δ is the time step of the discretization and $a = \exp\left(-\frac{\Delta}{\tau}\right)$. This is the behavior of a recursive or Infinite Impulse Response (IIR) filter.

Due to the discrete time approximation, the input spike times need also to be discretized. Instead of a sum of Dirac pulses, the input spike trains are now modeled by sums of Kronecker deltas $\delta_{i,j}$.

For digital implementations, we do not only have to discretize time, but also all numbers. In this work we assume that numbers are represented using two's complement, to ensure a better unification of the different techniques. We will not present a study on the quantization and precision of all variables because they have already extensively been studied in [57]. There is was shown that floating point precision is not needed for SNN to perform tasks. Even a word width of 5 to 7 bits for the important parameters was enough for the network to perform its task. In this work we assume a word width of 10 bits for synapse model potentials, membrane potentials and all weights.

The above mentioned recursive equation for performing exponential decay is usually implemented using an explicit multiplication with a constant smaller, but close to 1. In hardware, a full-fledged multiplication unit requires a significant amount of silicon real-estate and are power inefficient. Since low-cost FPGA don't have dedicated multiplication hardware, and in high end FPGA's, the number of multipliers is much smaller than the number of neurons that can be implemented in a chip, we will investigate other technique for implementing this decay using very limited hardware resources.

There are two ways to approximate exponential decay without using hardware multiplications: by using linear decay or shift operations.

Linear Decay When linear decay is used to approximate exponential decay, we must choose the linear decay rate according to the time constant of the exponential decay. Many options are possible (first derivative of exponential, same time until full decay, ...) but we chose to preserve the integral of the decayed function. This is important because the integral of a decayed spike at a synapse represents the total charge that is injected into the membrane. To have a similar membrane effect, this charge must be equal for the original exponential decay and for its linear approximation. The integral of an exponentially decaying unit spike is equal to

$$\int_0^\infty \exp\left(\frac{-t}{\tau}\right) \mathrm{d}t = \tau \tag{7}$$

while the integral of a linear decay is equal to

$$\int_0^\infty \max\left(0, 1 - bt\right) \mathrm{d}t = \frac{1}{2b}. \tag{8}$$

Both integrals are equal if we set the linear decay factor to $b = 1/2\tau$. Note that linearly decaying both the synapse model and the membrane results in a PSP that has a quadratically increasing part followed by a linearly decaying part.

Binary Shifts Without much more extra additions than linear decay, exponential decay can be well approximated through binary shift operations. This is in fact just a simplified binary multiplication with one fixed operand. In what follows we assume the binary point to be left of the most-significant bit. The basic idea of binary shifts is that the decay factor a is approximated by $\sum_{k\in M} 2^{-k}$ or $1 - \sum_{k\in N} 2^{-k}$ where sets M and N are the binary shift values. The choice between both options is based on the number elements in both sets because this determines the number of additions that need to be done. Because $a = \exp\left(-\frac{\Delta}{\tau}\right)$ and τ is normally approximately one or two magnitudes larger than Δ, a is only a bit smaller than 1. The most economic way to represent this using binary shifts is to use $1 - \sum_{k\in N} 2^{-k}$. The actual binary implementation of this is $y - \sum_{k\in N} y \gg k$ which in two's complement is equal to $y + \mathrm{not}\left(\sum_{k\in N} y \gg k\right) + 1$. This operation can be implemented in hardware very space-efficiently. There is a little catch with the binary shift approximation: because the binary shift operation actually also performs a floor round-off, the decay operation does not decay completely to zero, but stop at $2^{-\min(N)}$ because then the shift operation gets completely rounded off (this is only the case for positive numbers, negative numbers do decay towards zero) [7]. This can be corrected by removing the $+1$ whenever y is positive and one of the bit positions in the range $[\min(N) - 1, 0]$ is

[7] Depending on the implementation this can also be the other way round.

one which is easy to implement. Note however that this makes the positive decay a bit non-homogeneous and faster than the negative decay.

6 Overview of possible hardware implementations

We will now give an overview of the different possibilities to implement a SNN processor into digital hardware. This is split into several parts: how is the neuron PE implemented and how are the PEs interconnected.

6.1 Neuron architecture

When processing LIF neurons, two main operations are needed: accumulating all active input weights in the synapse models and/or membrane, and performing the decay inside the synapse models and membrane. The number of inputs to a neuron is weakly dependant on the network size and usually ranges from 10 to 1000, while the total number of synapse models is kept quite small to limit hardware usage (1 to 10), so adding-up the weights is one of the most performed operations. This can be done in two ways, serially or in parallel:

- **Serial processing.** The addition can be done serially by using an accumulator. The processing of the membrane and synapse models can also be implemented on the same accumulator by using multiple registers.
- **Parallel processing.** Here, the calculation of all the incoming weight additions is done in parallel by implementing an adder tree in hardware. To improve performance, it is also beneficial to pipeline[8] the implementation by adding flip-flops after each adder. It is advised to also implement the membrane and synapse models in parallel to avoid a bottleneck.

The arithmetic used to calculate the spiking neurons will also have a large effect on the size and speed of the implementation. Three types of arithmetic are possible:

- **Serial arithmetic.** The addition and shifting of bit vectors can easily be implemented in a serial fashion. Size is reduced, but speed decreases.
- **Parallel arithmetic.** All operations are done on multi-bit vectors, like on classic processors. This implementation option allows for fast operation, but is larger in size.

[8] Pipelining is a hardware optimization technique that adds registers inside long combinatorial paths to shorten the delay introduced by these long paths. This operation allows to increase the clock speed and throughput but adds latency and added control complexity by the need for pipeline initialization and run-out.

6.2 Interconnection architecture

Interconnection of the PEs can also be done in several ways:

- **Direct interconnection.** All PEs are directly connected via wires. For relatively small (¡ 1000) fixed and sparse networks this is an easy, compact and fast option. Time-sharing of neuron PEs is only possible if connections are also shared. Reconfiguration of the interconnection is only possible by resynthesis and reconfiguration of the hardware.
- **Memory based interconnection.** Communication is handled completely by one or more memories. Time-sharing and reconfiguring the neurons and interconnections is now easy. The major drawback is that the memory bandwidth becomes a problem when network size increases.
- **Bus based interconnection.** When using one or more buses to communicate information between the neurons in a network, a certain amount of extra logic is needed in each of the PEs. The bus also quickly becomes a bottleneck. Time-sharing and reconfiguring the interconnection is possible though not simple. Spike oriented buses like Address Event Representation, pulse stream and non-arbitrated pulse frequency modulation (see Chapter 7 in [58] for an overview of these techniques) can alleviate some of these problems but is still quite complex to implement.

7 Existing compact hardware architectures for SNNs

As was pointed out in the introduction, already various implementation architectures for SNNs have been investigated. In this section we briefly discuss two previously published architectures which we re-implemented. The first is an approach inspired on classic processor design where processing is serial and arithmetic is parallel; the second approach takes advantage of several FPGA specific properties: it uses distributed memory, parallel processing but with serial arithmetic to limit size. We added synapse models and various optimisations. The general and speech recognition application specific hardware usage are given.

7.1 Serial Processing, Parallel Arithmetic

This architecture processes the neurons as would be done on a classic CPU: by serially processing all synapses, synapse models and membrane using parallel arithmetic (SPPA). The SNN hardware presented in [30] and [7] are similar, but our implementation incorporates exponential synapse models, different

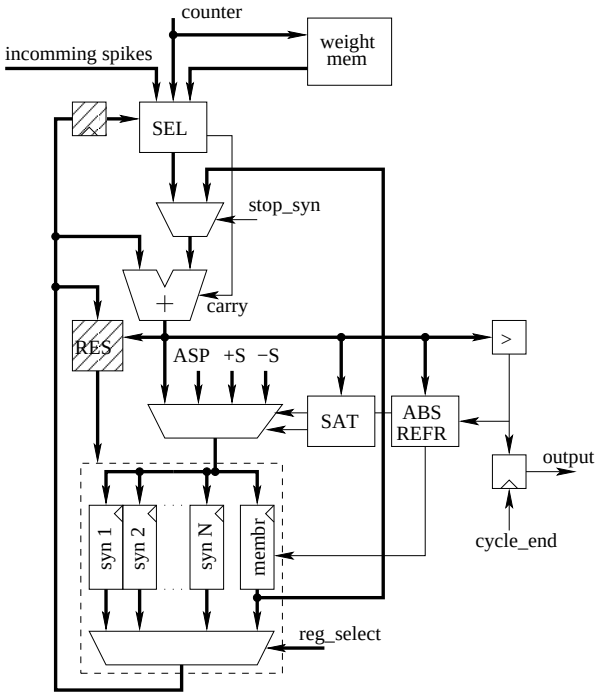

Fig. 3. SPPA architecture. Thick lines represent multi-bit signals, dashed boxes represent optional structures.

decay options and saturation; and occupies less hardware, allowing larger networks on the same FPGA. An overview of the architecture is given in Figure 3. Each PE processes only one neuron (direct mapping) and the PEs are directly interconnected.

The SNN is simulated using time step based simulation. Each time step consists of several operations, one per clock tick. These operations are: adding weights to the membrane or synapse accumulator, adding the synapse accumulators to the membrane's and decaying these accumulators. Threshold detection and membrane reset is performed in parallel with these operations. Weights are stored in memory which is located in each PE, while the accumulators are stored in a register bank. Several options for decay approximation and refractoriness are supported by this implementation. They can be easily selected by adding a configuratio n file to the VHDL hardware description. When a regular network structure is used (all PEs have the same number of inputs, where some may be unconnected), a single controller can be used to steer the complete network.

When implementing this architecture in FPGA we can derive general (but approximate) scaling formulas for area, memory and time usage with respect to the main network parameters. These are presented in the top row of Table **??**. The controller uses approximately 3B 4-LUTs. We used binary shift exponential decay approximation and a combination of absolute and relative refractoriness. For the actual 200 neuron reservoir that is used for the speech

Table 2
SPPA hardware results. The parameters for the general formulas are: N number of neurons, I number of inputs per neuron, B internally used word width, S the number of distinct synapse time constants, and T the number of 'steps' used in approximating exponential decay of the synapses and membrane (this is usually 1 of 2).

	Properties			
	4-LUTs	FFs	RAM	Cycles
General	$N\,(I/2+2B+10)$	$N\,(B(S+1)+1)$	$BN(ST+I)$	$(T+1)\,(S+1)+I$
Speech task	$13,812$	$4,038$	$56 \times 16,384$	18

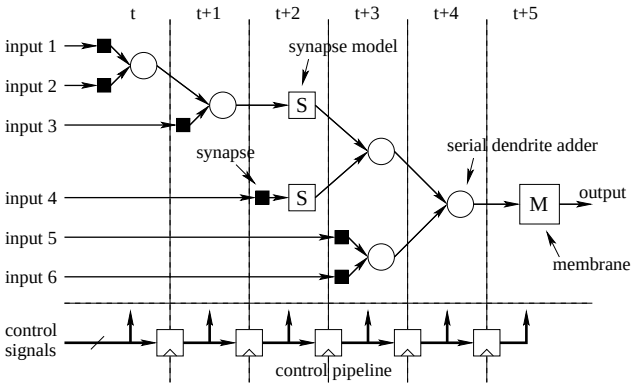

Fig. 4. Overview of the PPSA neuron architecture

recognition application, the implementation results are shown in the lower row (area optimization was turned on for all designs). Note that 56 standard FPGA RAM block of each 16 kbit are used, but they are only partially filled. This design can be clocked at 100 MHz on state-of-the-art FPGAs (Xilinx's XC4VSX35). Given that the incomming speech is sampled at 16 kHz (which is also the rate at which the network operates) we get a network that is 347 times faster than needed for real-time processing. The FPGA is filled for 60 procent.

7.2 Parallel Processing, Serial Arithmetic

This architecture [46,33] is based on a parallel synapse processing scheme using serial arithmetic (PPSA) to limit the total size of the PE. Each PE only computes one neuron (direct mapping) and the different PEs are interconnected via direct, fixed connections. Because of the parallel processing scheme, all synaptic weights are stored in a parallel and decentralized way by using many very small memories (we actually use one memory per synapse). This is possible by using FPGA specific features (we use the Xilinx specific SRL16

Table 3
PPSA hardware results

	Properties			
	4-LUTs/SRL16s	FFs	RAM	Cycles
General	$N(22+3I+10S)$	$N(2I+BS+B)$	0	$\lceil \log_2(I) \rceil + B(1+T)+1$
Speech task	13,426+2,401	21,743	0	35

memories which are 16 bit shift registers that can be implemented in a single 4-LUT).

The parallel processing of the dendritic tree is performed by a binary, direct mapped adder tree where each adder is a one bit serial adder. To improve processing speed we use pipelining [9] . An overview of this architecture can be seen in Figure 4. For more details we refer to [46].

During each SNN time step, all inputs to the dendritic trees are presented in parallel. It takes $\lceil \log_2(inputs) \rceil$ cycles before the first bit is available at the end of the adder tree. Serially adding this result to the membrane accumulator is performed at one bit per cycle. After this, the membrane is decayed which again is implemented serially and thus takes one bit per clock cycle.

The hardware results for this architecture are presented in Table 3. The top row shows the general scaling properties, while the bottom row gives the result specific for the 200 neuron network used for the speech recognition application. The controller uses approximately 3B 4-LUTs. This design can be clocked at 115 MHz on the same FPGA and it is filled for 60 percent.. Each simulation time step takes 35 cycles which results in an implementation that is 205 times faster than real-time (with 16 kHz input). Although this design does parallel processing of the dendritic tree, it is slower than SPPA due to the limited number of input. With limited inputs the cost for doing serial arithmetic outweighs the gain made by the parallel dendritic tree. When more inputs are used (approximately 50) PPSA becomes faster than SPPA.

7.3 Serial Processing, Serial Arithmetic

Because both of the previously published architectures give much faster than real-time performance on the speech recognition task, they use more hardware than needed. We will now present a novel architecture for the processing of SNN that allow slower but scalable operations at a highly reduced hard-

[9] After each tree operation, memory elements are added. This allows higher clock speed, but requires multiple clock cycles before the result is available at the output.

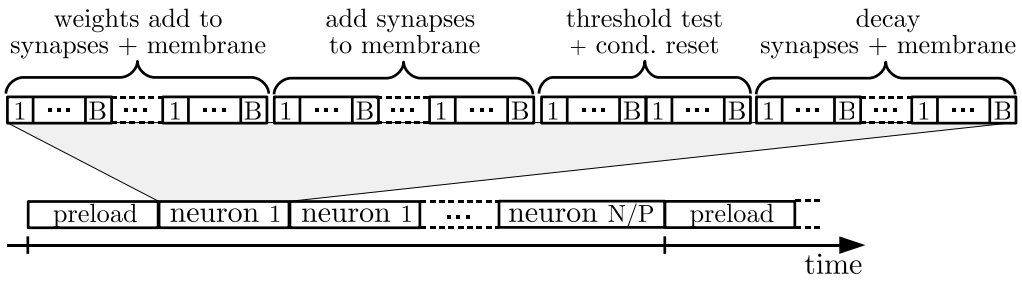

Fig. 5. SPSA timing

ware cost. The architecture processes all synapses serially as well as doing all arithmetic serially (SPSA). This results in a very small implementation of the PE (only 4 4-LUTs!) but in longer computing times. All neuron information is stored in RAM and interconnection between several neurons is also memory based. Due to this, each PE can process several neurons serially, but this at the cost of speed. Because the controller is much larger than the actual PE, we will opt for using one controller and several PEs. To do this, all PEs have to perform the same instructions on different data. This is called a Single Instruction/Multiple Data (SIMD) architecture (the SNN FPGA implementation presented in [32] is also a SIMD processor but it uses parallel arithmetic). Processing the network consists of two phases: updating the neuron states and performing the interconnection. We chose to do both operations in parallel (unlike [32] which uses two separate phases).

The general setup is that we use one controller, several PEs and each PE processes several neurons per simulated time-step. We will call the time needed to simulate a single neuron a cycle. Each simulated time-step thus consists of several cycles. A simplified control flow is shown in Figure 5. The global cycle structure of weight adding, decay and threshold testing is quite similar to the other architectures, but now each of these command is processed serially, i.e. that they consist of several clock cycles.

An overview of the system is given in Figure 6a. Several processing elements connect parallelly to 4 memories where each memory gets an address from the controller. The synapse and membrane memory is the working memory of the PEs holding the synapse and membrane accumulators (and some constants). Note that for each PE this is actually a one bit memory, but because all PEs get the same instructions the addresses to these memories are the same and can thus be implemented as one multi-bit memory. The second memory is a circular buffer holding the weights. Note that the weight a placed in serially with several weights in parallel for the different PEs. The third and fourth memories are used for input and output spikes to and from the PEs. These are actually double buffered so that the interconnection PE can copy data while the neuron PEs are running. The interconnection PE is organized in such a way that receiver-oriented copying is performed: at cycle T, the

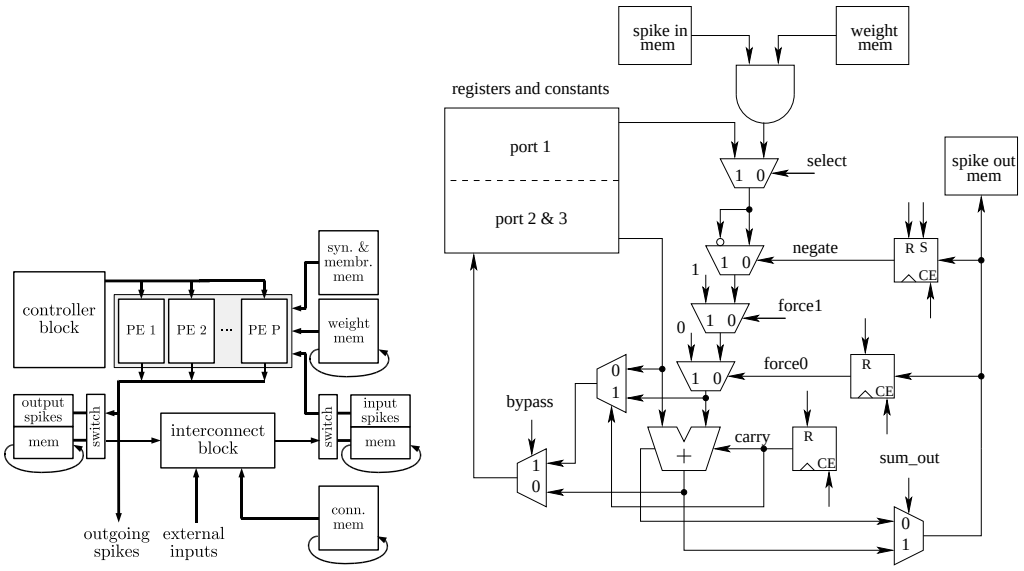

Fig. 6. SPSA overview and PE internals

interconnection PE is copying the input spikes for the neurons processed in cycle $T+1$ into the second buffer of the input spike memory. At the next cycle, these two memories are switched. Spike output memory is switched after each complete simulated time-step. External input and output is possible via a memory interface. Note that all spike copying is done on a bit by bit basis. Both the input and output spike memories are two-port memories which have a one-bit and a multi-bit port.

The actual neuron PE data-pad can be seen in Figure 6b. Although it might seem quite complex, it can fit in 4 4-LUTs. The PE uses a 3-port memory [10] which hold the membrane and synapse model potential, and all constants such as decay factors, threshold and absolute refractory. The data path of the PE is centered around a one-bit adder. One port is directly connected to the adder while the other adder input can be a weight or one of the ports of the 3-port memory, which can then be negated, forced to one or forced to zero. The output of the adder is fed to a bypass multiplexer to allow the inputs to bypass the adder. The result is written back to the 3-port memory. Some of the control signals are directly generated by the controller while others a mediated through a flip-flop which has a clock enable and a synchronous set and reset. The actual control signals and their timing behavior will not be elaborated on for brevity.

An overview of the approximate size results for the neuron PE, the controller and the interconnection is given in Table 4, where P stands for the number

[10] Three-port memory is not a primitive component of FPGAs but can easily be emulated thanks to the high speed memories (memories are 5 times faster than our actual design). This allows the memory to run at double speed and convert a two-port memory into a four-port memory by time-multiplexing.

Table 4
SPSA general size results

	Properties			
	Number	4-LUTs	FFs	RAM
PE	P	4	3	$B(2S+6)+2IP+2N$
Controller	1	270	$33+2\log_2(\lceil N/P \rceil BI)$	0
Interconnect	1	170	$\log_2(NSP\lceil N/P \rceil^2)$	$IN\log_2(N\lceil N/P \rceil)$

Table 5
SPSA speech application hardware results

	4-LUTs			Clock		Times
PEs	LUT-RAMs	FFs	RAM	cycles	Mhz	real-time
1	488+40	223	9×16,384	38994	145	0.23
5	489+40	223	9×16,384	7954	145	1.1
10	489+40	223	9×16,384	4074	143	2.2

of PEs. The number of clock ticks per cycle is equal to $(S+3+B(I+2S+5))(\lceil N/P \rceil+1)$. The first part is the number of clock ticks needed per cycle, while the second part is the number of cycles. Notice that one cycle is added to allow pre-loading by the interconnection block of the input spikes for the first cycle.

The results for the speech application are summarized in Table 5. Note that some of the memory is implemented in LUTs using LUT-RAM, the column denoting 'clock cycles' represents the number of clock cycle per simulated time-step, and 'real-time' denotes how much faster than real-time processing that option is. We tested an increasing number of PEs which results in a large final speed variation, however with very little space variation. Optimal settings for real-time processing are achieved with 5 PEs.

With the current hardware we can maximally and in real-time process approximately 1600 neurons at 16 kHz and this using 40 PEs. When processing larger networks at this speed, the interconnect block becomes a bottleneck.

Note that adding more classes to the classification problem (for example if going from isolated digits to phonemes), the architecture stays exactly the same, only more readout functions need to be calculated. The reservoir implementation and the number of nodes that are read-out stay exactly the same.

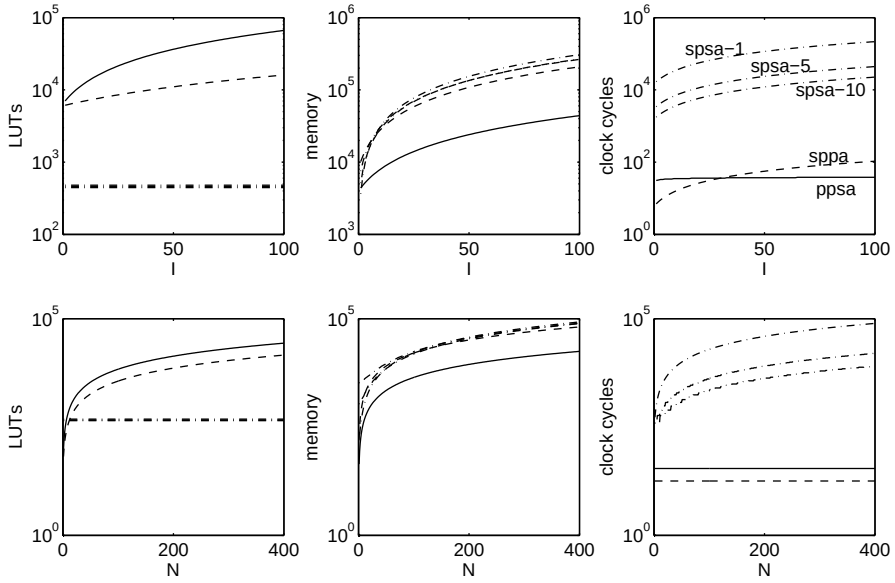

Fig. 7. Comparison of the three architectures (the SPSA architecture with 1, 5 and 10 PEs) with respect to the number of inputs and the number of neurons.

8 Comparison

The three different architectures have very different area, memory and time scaling properties. An approximation of the requirements for the three different designs is given in Tables 2, 3 and 4. To compare the designs, Figure 8 shows the number of 4-LUTs per PE, the memory (RAM and FF) usage and the number of clock cycles per time step needed for each of the three architectures with respect to the number of inputs and the number of neurons. The others settings are $I = 12, N = 200, B = 10, S = 1, T = 2$. The number P of PEs for the SPSA case is set to 1, 5 and 10. Notice the logarithmic scale of the y-axis.

When increasing the number of inputs, LUT usage stays constant for SPSA, but the number of clock cycles needed per time step increases linearly and is significantly higher than the other architectures. Memory usage is quite high, but LUT usage is very low. Notice that increasing the number of PEs has a big influence on the number of clock cycle, but almost non on area and memory. Similar results are acchieved when increasing the number of neurons. When increasing the number of neurons, in the case of SPSA, the computing time increasing because PEs are time-shared, while the other architectures have fixed computing time, since neurons are directly mapped to dedicated PEs. For SPPA, the memory requirement is quite high, but LUT usage and computing time are limited, while PPSA uses much hardware, not much memory is needed (because LUTs are used as memories) and operation is fast. Notice however that when increasing the number of inputs, for a small number of inputs,

SPPA is actually faster than PPSA. This is due to the pipeline structure that is used in PPSA which needs warm-up and cool-down after each cycle. The fastest solution is thus application dependent. Many applications also have area, power or real-time constraints. Using these different implementations, we are now able to make a trade-off between the important properties of the different architectures, and choose the one that best fits the constraints of the application. We will demonstrate this in the next section, where we will implement a LSM based speech recognition system using the three presented architectures.

9 System architecture

10 Conclusions and future work

In this work we showed that real-time speech recognition is possible on limited FPGA hardware using an LSM. To attain this we first explored existing hardware architectures (which we reimplemented and improved) for compact implementation of SNNs. These designs are however more than 200 times faster than real-time which is not desired because lots of hardware resources are spend on speed that is not needed. We present a novel hardware architecture based on serial processing of dendritic trees using serial arithmetic. It easily and compactly allows a scalable number of PEs to process larger networks in parallel. Using a hardware oriented RC design flow we were able to easily port the existing speech recognition application to the actual quantized hardware architecture.

For future work we plan to investigate different applications, such as autonomous robot control, large vocabulary speech recognition, and medical signal processing, that all use the hardware LSM architectures presented in this work, but which all have very different area/speed trade-offs. Parameter changing without resynthesis will also be investigated (dynamic reconfiguration or parameter pre-run shift-in with a long scan-chain are possibilities).

References

[1] W. Maass, Noisy spiking neurons with temporal coding have more computational power than sigmoidal neurons, in: M. Mozer, M. I. Jordan, T. Petsche (Eds.), Advances in Neural Information Processing Systems, Vol. 9, MIT Press (Cambridge), 1997, pp. 211–217.

[2] O. Booij, Temporal pattern classification using spiking neural networks, Master's thesis, University of Amsterdam (2004).

[3] B. Schrauwen, J. Van Campenhout, Backpropagation for population-temporal coded spiking neural networks, in: Proceedings of the Internation Joint Conference on Neural Networks, 2006, pp. 1797–1804.

[4] A. Delorme, S. Thorpe, Face identification using one spike per neuron: resistance to image degradations, Neural Networks (2001) 795–804.

[5] D. Verstraeten, B. Schrauwen, D. Stroobandt, J. Van Campenhout, Isolated word recognition with the liquid state machine: a case study, Information Processing Letters 95 (6) (2005) 521–528.

[6] D. Verstraeten, Een studie van de Liquid State Machine: een woordherkenner, Master's thesis, Ghent University, ELIS department (2004).

[7] D. Roggen, S. Hofmann, Y. Thoma, D. Floreano, Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot, in: NASA/DoD Conference on Evolvable Hardware, 2003, pp. 189–198.

[8] D. Floreano, J. Zufferey, J. Nicoud, From wheels to wings with evolutionary spiking neurons, Artificial Life 11 (1-2) (2004) 121–138.

[9] S. M. Bohte, H. L. Poutré, J. N. Kok, Error-backpropagation in temporally encoded networks of spiking neurons, Neurocomputing 48 (1–4) (2002) 17–37, a short version of this paper has been published in the proceedings of ESANN'2000.

[10] B. Schrauwen, J. Van Campenhout, Extending SpikeProp, in: Proceeding of the International Joint Conference on Neural Networks, 2004, pp. 471–476.

[11] T. Natschläger, W. Maass, Spiking neurons and the induction of finite state machines, Theoretical Computer Science: Special Issue on Natural Computing 287 (2002) 251–265.

[12] H. Jaeger, H. Haas, Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless telecommunication, Science 308 (2004) 78–80.

[13] D. Verstraeten, B. Schrauwen, M. D'Haene, D. Stroobandt, A unifying comparison of reservoir computing methods, Neural Networks 20 (2007) 391–403.

[14] D. Anguita, M. Valle, Perspectives on dedicated hardware implementations, in: Proceedings of the European Symposium on Artificial Neural Networks, 2001, pp. 45–56.

[15] I. Aybay, S. Cetinkaya, U. Halici, Classification of neural network hardware, in: Neural Network World, Vol. 6, 1996, pp. 11–29.

[16] J. B. Burr, Digital neural network implementations., in: P. Antognetti, V. Milutinovic (Eds.), Neural Networks: Concepts, Applications, and Implementations, Volume 2, Prentice Hall, 1991, pp. 237–285.

[17] P. Moerland, E. Fiesler, Neural network adaptations to hardware implementations., in: E. Fiesler, R. Beale (Eds.), Handbook of Neural Computation, Institute of Physics Publishing and Oxford University Publishing, New York, 1997, pp. E1.2:1–13, iDIAP-RR 97-17.

[18] T. Schoenauer, A. Jahnke, U. Roth, H. Klar, Digital neurohardware: Principles and perspectives, in: Neural Networks in Applications, 1998, pp. 101–106.

[19] J. Zhu, P. Sutton, FPGA implementation of neural networks - a survey of a decade of progress, in: P. Y. Cheung, G. Constantinides, J. de Sousa (Eds.), Proceeding of the 13th International Conference on Field-Programmable Logic and Applications, Lisbon, Portugal, 2003, pp. 1062–1066.

[20] B. Girau, Neural networks on FPGAs: a survey, in: Proceeding of Second ICSC Symposium on Neural Computation, 2000.

[21] A. R. Omondi, J. C. Rajapakse (Eds.), FPGA Implementation of Neural Networks, Springer, 2006.

[22] A. Jahnke, U. Roth, H. Klar, Towards efficient hardware for spike-processing neural networks, in: Proceedings of the World Congress on Neural Networks, 1995, pp. 460–463.

[23] A. Jahnke, T. Schoenauer, U. Roth, K. Mohraz, H. Klar, Simulation of spiking neural networks on different hardware platforms, in: Proceedings of the International Conference on Artificial Neural Networks, Springer Verlag Berlin, 1997, pp. 1187–1192.

[24] R. Preis, K. Salzwedel, C. Wolff, G. Hartmann, Efficient parallel simulation of pulse-coded neural networks (PCNN), in: Proceedings of the PDPTA 2001, Vol. 1, Las Vegas, Nevada, USA, 2001, pp. 463–470.

[25] M. Schäfer, T. Schönauer, C. Wolff, G. Hartmann, H. Klar, U. Rückert, Simulation of spiking neural networks - architectures and implementations, Neurocomputing 48 (1-4) (2002) 647–679.

[26] L. S. Smith, A. Hamilton (Eds.), Neuromorphic Systems: Engineering Silicon from Neurobiology, World Scientific, 1998.

[27] W. R. Softky, Fine analog coding minimizes information transmission, Neural Networks 9 (1) (1996) 15–24.

[28] D. Floreano, N. Schoeni, G. Caprari, J. Blynel, Evolutionary bits'n'spikes, in: R. K. Standish, M. A. Beadau, H. A. Abbass (Eds.), Artificial Life VIII: Proceedings of the Eight International Conference on Artificial Life, MIT Press, 2002, pp. 335–344.

[29] J. Nielsen, H. H. Lund, Spiking neural building block robot with hebbian learning, in: Proceedings of IEEE International Conference on Intelligent Robots and Systems, IEEE Press, Las Vegas, Nevada, 2003, pp. 1363–1369.

[30] A. Upegui, C. A. Peña Reyes, E. Sanchez, An FPGA platform for on-line topology exploration of spiking neural networks, Microprocessors and Microsystems 29 (2005) 211–223.

[31] S. Bellis, K. M. Razeeb, C. Saha, K. Delaney, C. O'Mathuna, A. Pounds-Cornish, G. de Souza, M. Colley, H. Hagras, G. Clarke, V. Callaghan, C. Argyropoulos, C. Karistianos, G. Nikiforidis, FPGA implementation of spiking neural networks - an initial step towards building tangible collaborative autonomous agents, in: FPT'04, International Conference on Field-Programmable Technology, Brisbane, Australia, 2004, pp. 449–452.

[32] M. J. Pearson, C. Melhuish, A. G. Pipe, M. Nibouche, I. Gilhesphy, K. Gurney, B. Mitchinson, Design and FPGA implementation of an embedded real-time biologically plausible spiking neural network processor, in: Proceeding of the 15th International Conference on Field Programmable Logic and Applications, FPL, 2005, pp. 582–585.

[33] B. Girau, C. Torres-Huitzil, FPGA implementation of an integrate-and-fire LEGION model for image segmentation, in: Proceeding of the 14th European Symposium on Artificial Neural Networks, 2006, pp. 173–178.

[34] A. Jahnke, U. Roth, H. Klar, A SIMD/dataflow architecture for a neurocomputer for spike-processing neural networks (NESPINN), in: 5th International Conference on Microelectronics for Neural Networks and Fuzzy Systems (MicroNeuro'96), Lausanne (Switzerland), 1996, pp. 232–237.

[35] T. Schoenauer, N. Mehrtash, A. Jahnke, H. Klar, MASPINN: Novel concepts for a neuro-accelerator for spiking neural networks, in: VIDYNN'98 – Workshop on Virtual Intelligence and Dynamic Neural Networks, Stockholm, 1998, pp. 87–96.

[36] N. Mehrtash, D. Jung, H. Hellmich, T. Schoenauer, V. Lu, H. Klar, Synaptic plasticity in spiking neural networks (SP^2INN): A systems approach, IEEE Transactions on Neural Networks 14 (5) (2003) 980–992.

[37] H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, H. Klar, Emulation engine for spiking neurons and adaptive synaptic weights, in: Proceedings of International Joint Conference on Neural Networks, Montreal, Canada, 2005, pp. 3261–3266.

[38] G. Hartmann, G. Frank, M. Schäfer, C. Wolff, SPIKE128K - an accelerator for dynamic simulation of large pulse-coded networks, in: Proceedings of the 6th MicroNeuro, 1997, pp. 130–139.

[39] E. Ros, E. Ortigosa, R. Agís, R. Carrillo, A. Prieto, M. Arnold, Spiking neurons computing platform, in: Proceedings of the 8th International Work-Conference on Artificial Neural Networks, IWANN 2005, 2005, pp. 471–478.

[40] J. Waldemark, M. Millberg, T. Lindblad, K. Waldemark, V. Becanovic, Implementation of a pulse coupled neural network in FPGA, International Journal of Neural Systems 10 (3) (2000) 171–177, special Issue on Hardware Implementations.

[41] C. Grassmann, J. Anlauf, RACER - a rapid prototyping accelerator for pulsed neural networks, in: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02), 2002, pp. 277–278.

[42] H. de Garis, M. Korkin, F. Gers, E. Nawa, M. Hough, Building an artificial brain using an FPGA based CAM-Brain Machine, Applied Mathematics and Computation 111 (2–3) (2000) 163–192.

[43] B. Glackin, T. M. McGinnity, L. P. Maguire, Q. X. Wu, A. Belatreche, A noval approach for the implementation of large scale spiking neural networks on FPGA hardware, in: Proceedings of the 8th International Work-Conference on Artificial neural Networks, IWANN 2005, 2005, pp. 552–563.

[44] F. Schürmann, K. Meier, J. Schemmel, Edge of chaos computation in mixed-mode VLSI - "A hard liquid", in: L. K. Saul, Y. Weiss, L. Bottou (Eds.), Advances in Neural Information Processing Systems 17, MIT Press, Cambridge, MA, 2005, pp. 1201–1208.

[45] D. Verstraeten, B. Schrauwen, D. Stroobandt, Reservoir-based techniques for speech recognition, in: Proceedings of the World Conference on Computational Intelligence, 2006, pp. 1050–1053.

[46] B. Schrauwen, J. Van Campenhout, Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic, in: Proceedings of the European Symposium on Neural Networks, 2006, pp. 623–628.

[47] W. Maass, Networks of spiking neurons: the third generation of neural network models, Neural Networks 10 (9) (1997) 1659–1671.

[48] R. Lyon, A computational model of filtering, detection and compression in the cochlea, in: Proceedings of the IEEE ICASSP, 1982, pp. 1282–1285.

[49] B. Schrauwen, J. Van Campenhout, BSA, a fast and accurate spike train encoding scheme, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN), 2003, pp. 2825–2830.

[50] R. Legenstein, W. Maass, New Directions in Statistical Signal Processing: From Systems to Brain, MIT Press, 2005, Ch. What makes a dynamical system computationally powerful?

[51] H. Jaeger, Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the "echo state network" approach, Tech. Rep. GMD Report 159, German National Research Center for Information Technology (2002).

[52] E. Izhikevich, Which model to use for cortical spiking neurons?, IEEE Transactions on Neural Networks 15 (2004) 1063–1070.

[53] W. Gerstner, W. Kistler, Spiking Neuron Models, Cambridge University Press, 2002.

[54] H. Markram, J. Lübke, M. Frotscher, B. Sakmann, Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs, Science 275 (5297) (1997) 213–215.

[55] W. Maass, A. Zador, Computing and learning with dynamic synapses, in: W. Maass, C. Bishop (Eds.), Pulsed Neural Networks, The MIT Press, 1999, pp. 321–336.

[56] J. Triesch, Synergies between intrinsic and synaptic plasticity in individual model neurons, in: Advances in Neural Information Processing Systems (NIPS 2004), Vol. 17, 2004.

[57] U. Roth, A. Jahnke, H. Klar, Hardware requirements for spike-processing neural networks, in: Proceeding of the International Work-Conference on Artificial Neural Networks, 1995, pp. 720–727.

[58] W. Maass, C. Bishop, Pulsed Neural Networks, Bradford Books/MIT Press, Cambridge, MA, 2001.